The Report Committee for Samuel Teklu Cherinet
Certifies that this is the approved version of the following report:

Intelligent Vision Marketplace: Feasibility Study

APPROVED BY
SUPERVISING COMMITTEE:

Constantine Caramanis, Supervisor

Bijay Kusle

Intelligent Vision Marketplace: Feasibility Study

by

Samuel Teklu Cherinet

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2020

Dedication

This is dedicated to my lovely wife, without whom I wouldn’t have been able to start this journey at all, or made this far.

Abstract

Intelligent Vision Marketplace: Feasibility Study

Samuel Teklu Cherinet, M.S.E

The University of Texas at Austin, 2020

Supervisor: Constantine Caramanis

Recent years have been very generous for machine learning, as we have seen a tremendous growth in that industry. And this advancement can be further boosted by making it easily accessible to the general public. The goal of this report is to explore if the infrastructure and tooling available today would allow for such a platform to exist. This report looks into currently available image/video processing applications and dive into the latest and greatest tool available. The outcome of this study will be a prototype of this marketplace along with my findings on availability of polished tools and an insight into the performance expectation and cost of running this platform.

Table of Contents

List of Tables

List of Figures

INTRODUCTION

Taking the courses that are directly related to machine learning and data mining in my master's course study was one of the most eye-opening experiences in my life. It was quite remarkable what can be done with machine learning, especially when it comes to computer vision. It was overwhelming to say the least, from a learning standpoint, but there was one thing that was bothering me throughout; why do I get the impression that the implementation is too complex and it is sparsely used in the real world?

We have made so much progress with applying machine learning to image recognition. But I felt we have not yet scratched the surface when it comes to actually letting the general public use it in their daily activity. And although the out of the box tools to train a model to perform simple image processing has improved drastically, the gap between a data scientist and an application developer still remains. So, what I started envisioning is a platform that is easy for both the general public and the professionals to use and create machine learning apps that are useful.

In the coming sections, in addition to just its simplicity, we will dive into the benefits of having a solid platform. Both from delivering the fruits of machine learning to the general public as well as creating an environment where users can in turn help fine tune models. This report will also explore the availability of tools to fulfill the goals stated above.

LITERATURE REVIEW

The premise of this report is , we lack many computer vision implementations as a front facing application because there is a gap between data scientists that are focused on making the best AI model to solve a problem and the app developers that are focused on delivering general purpose traditional apps to the general public.

App developers are reluctant to spend time to integrate machine learning in their application for many reasons. The learning curve is normally steep, although it has gotten better. A good example of this improvement is “Tensorflow Lite” for mobile applications, where you can use a trained model in your applications. The other reason for lack of traction is the possible inaccuracy that might make the implementation unreliable, which has a very detrimental impact on the quality of the app developers are trying to sell.

Before heading into solving this problem, it is worth taking a look at what is available today for the general public to use when it comes to Computer vision applications. One common implementation is the applications that detect friends and family members from the photos you have taken and suggest them for your approval. And there are those kinds of applications that are more hands on, where users access the application looking to fulfil a certain goal. An example of that is the direct deposit feature provided with the majority of banking apps freely available in app stores. There are also features provided by furniture stores on their apps to place their items in your living room using augmented reality. The Wayfair android app is a good example of that.

Figure 1: Wayfair mobile app implementation of computer vision to virtually place a couch in a room

These are just a few of the examples of machine learning implementations, particularly on visuals, in our day to day application consumption. Below are also implementations worth mentioning:

- Video Surveillance
- Visual Search for products using images
- Social media implementation of image and video filters that are wildly popular

And in recent years there is a big push to make machine learning tools as a whole (not just computer vision) readily available to a traditional application developer either as

PaaS (Platform as a service) or IaaS (Infrastructure as a Service). Here are few examples of these services available at the time of this report:

- AzureML: this is a Microsoft provided service that brings simplicity and scale to machine learning. It supports most of the commonly used tools and languages, and it comes with a Studio for a drag and drop.
- Amazon SageMaker: this is the Amazon (AWS) provided service that provides simple and scalable machine learning capability. It also comes with an IDE.
- AI Hub: this service is by Google, which provides similar service as the others are offering.
- Paperspace
- And all the Major cloud players provide virtual machine images that have all tools already installed and configured.

These services do ease the effort on doing research and having the tools available for data scientists to explore more ground. But there is still a gap between having the perfect AI model to address a real-life problem, and delivering it to the general public user for daily consumption. This is evident with not just image processing but with other disciplines of machine learning. There is still a steep learning curve for data scientists to take their pretrained model to a presentable format to the public. And the same challenge exists for the traditional developer to go to these services and try to figure out how to take his brilliant idea into a workable machine learning wizard.

The platform envisioned in this report attempts to provide a focused service that would allow data scientists to take their pretrained models and algorithms and directly serve the end user without worrying about building any user facing application.

IMPLEMENTATION

Inception

"Intelligent Vision Marketplace", which will be referred to as the platform from here on, is a digital storefront where computer vision applications are provided for consumption for the general public.

The diagram below provides a very high level visual on what components make up the platform and how they are linked with each other.

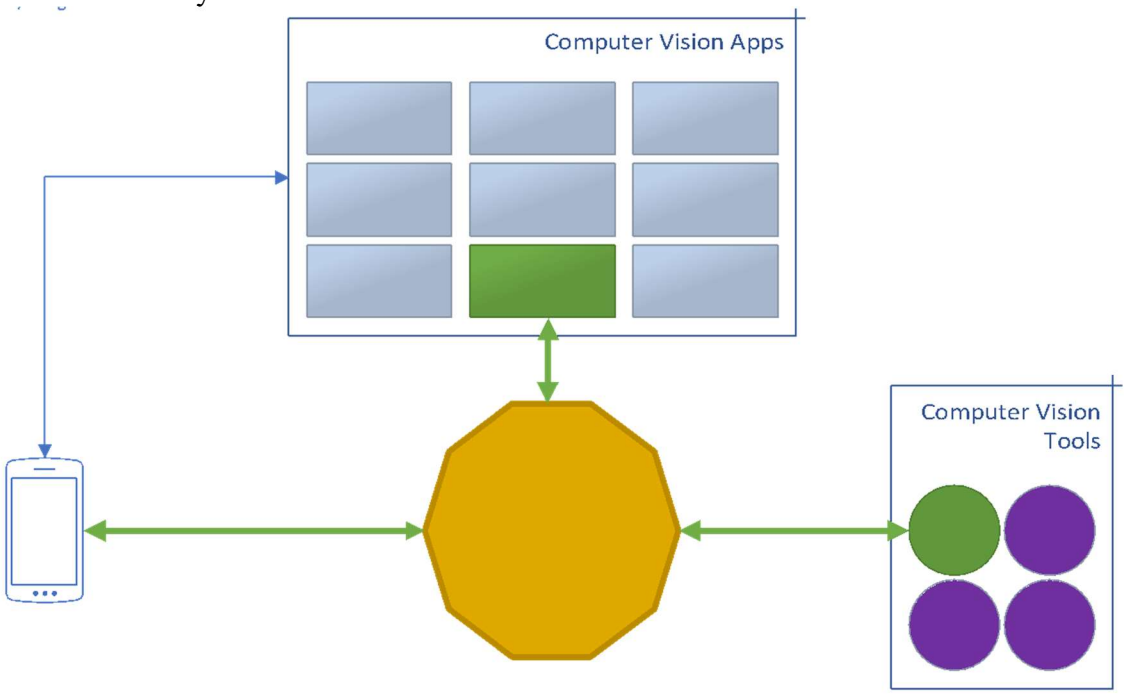


Figure 2: 10K-foot view of the platform

Using portals, either in a mobile device or in a desktop browser, users will have access to the repository of computer vision applications available for consumption. Once users select a computer vision application, they will be guided through providing all the required inputs needed to perform machine learning processing for that specific application.

The core, represented by the orange decagon, will pull the required computer vision tool based on the computer vision application that is selected into the runtime. And execute the application with the input from the portal as a parameter. Once the execution is completed, the portal will display the result back to the portal.

The next section will list out the features that will make this platform the most useful. Based on the nature of how they will be presented to a group of users, the application is divided into three major components; Consumer facing, Developer facing and Core. These will be all the features that will make the platform complete but, in the prototype, to support this report will be limited to the features that are essential to have a running service. The features described are better described by looking at the wireframes included in the Appendix this report.

Features

CONSUMER FACING

This part of the platform will provide a way for the general public to use the computer vision apps on a daily basis. It will be delivered in a web browser or mobile app, making it easy for users to provide images and photos. For example, a mobile app user will be able to install it on her/his device, and use it as a mobile camera addon. The wireframe in Figure 3 is how users will be able to browse the computer vision application repository.

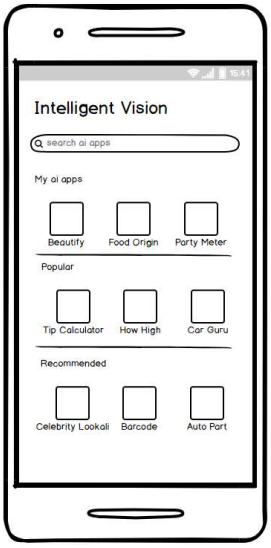


Figure 3: Proposed wireframe for browsing and searching computer vision applications.

Below is a summary of the features that will be available to the general consumer:

- Ability to browse, search and access computer vison applications
    - search application by name
    - search application by category
    - allow authenticated users to mark application as favorite
    - list recently used application
    - list recommended applications
    - list popular applications
- Ability to provide additional parameters to computer vision applications via a simple form UI
    - Image input from user
    - Text input from user
    - Number input from user
    - Date value from user

    - list item selection from user
- Ability to view results from AI analysis in a user-friendly format
- Ability to store input and results
    - display a list of recent runs by application
- Provide security to store inputs and results, both at rest and in transit.
- Ability to provide feedback (about accuracy) for computer vision applications
- Ability view multiple video feeds
- Ability to run analysis in the background as long as the feed is active
- Ability to set up triggers and notifications based on analysis.

DEVELOPER FACING

This portion of the platform will be used by developers that will create the computer vision apps for public consumption. This provides a protocol to allow developers to package their source code and models along with the needed parameters. And a protocol to display results to the user.

Normally developers would spend a vast amount of their time researching input data and trying to solve a problem using machine learning. They write thousands of lines of python code, spending hours after hours feature engineering, fine tuning parameters, and training their models for days. After they get an acceptable accuracy from their test data, now they are ready to give their fruits to the world. How this will work is, they will extract the output from their modeling and processing code and create a much smaller application that will just accept the desired input, and return a prediction of some sort. Below is a summary of the features that will be available to the developers:

- Allow developers to package and publish their apps

    - Provide application description
    - Provide expected input from users
    - Provide expected result from analysis and processing
    - Provide a template to display results to users.
    - Provide ability to version computer vision applications
- Allow developers to test apps before publishing
- Allow developers to monitor their app usage
- Allow developers to download feedback data from users to further fine tune their models/algorithm for better accuracy

CORE

The core, running user interface less for the most part, is what drives all the components to serve as the brain of the platform and will provided these functionalities

- Provide a protocol for developers to use to package their models, expected parameters, and result
- Run the analysis upon request / based on triggers
- Translate packages into runnable ML code
- Compile result

Design

Based on the requirements spelled out in the previous sections and the technical boundaries, technical architecture would look like this.

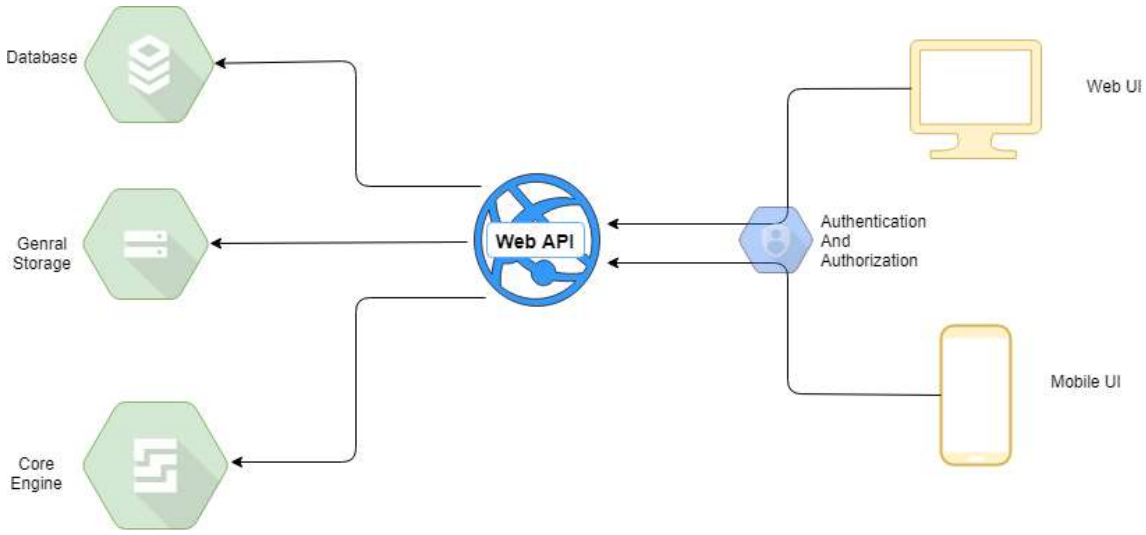


Figure 4: Technical Architecture

Database: This is a relational database storage that will support the application's need to store user, app, and app usage information.

General Storage: This is some sort of cloud storage where user inputs are securely kept. And it will be accessible by the core engine when users run apps in their context. The separation of this infrastructure from the web server as well as the engine will allow the platform to ensure the security and privacy of user's inputs from unauthorized access by the applications themselves.

Core Engine: The core engine is the heart of this platform, since this is "the component" that is responsible for running machine learning on inputs and delivering results to the end users.

Web API: To make the application structure more modularized and the integration technology agnostic, the platform utilizes restful web APIs. Communication between web UI and the server as well as the core engine is facilitated via these lightweight http calls.

Authentication and Authorization: Access to the platform will be secured by some form of authentication (traditional username password). And there will be a divide between

the role of the general public and developers based on their authorization. Only verified developers will be able to create and publish applications but it's use will be available for anybody with an account.

Computer Vision Application Package

As briefly introduced in the previous sections, this package is a single file or a collection of files that the core will be able to execute by passing user inputs and returning a result that is hopefully useful. it is worth noting here that this application package is not used to train a model but to utilize some out of the box machine learning tool to process an input and return a desired output.

The first question, and the most important one is, can we find these out of the box libraries (tools) that can be used to shelve a set of algorithms/models after the initial painfully long process of building the infrastructure? and the answer is yes. As mentioned earlier in this report, there are many tools today that have abstracted away all complexities of the math behind machine learning and allow developers to call few methods and get up to speed.

As an example, and as a guide through the design process, this report will use TensorFlow library to explore how It can be used to package and run as a computer vision application. For those who are not familiar, TensorFlow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications.

There are three major steps involved in creating an application for this platform, for that matter for any application that will be powered by machine learning as it's brain. The coming sections will walk through each step of the cycle.

DEVELOPMENT CYCLE

Step 1: Build the model

The first step before bundling an AI implementation into an app is actually processing a dataset, choosing the right algorithm, fine tuning parameters to get a prediction that is within an acceptable range of accuracy. This report doesn't really get into the details of how to train a model or to choose a library. Normally data scientists or enthusiasts will use something like google Colab or local Jupyter notebooks to work through the process of analysis and prediction.

To simulate this process, I took up the basic classification tutorial to build a simple model to get started "Classifying images of clothing". The code provided uses Fashion MINIST dataset to train a model in order to classify closing images into certain categories.

In addition to training the model, the main goal of this step is to store the knowledge collected from training the model using 70K images. TensorFlow provides a set of utility functions to save the entire trained model to load it later. Surprisingly the code was quite trivial, the builders of tensor have made this utility readily available to persist models to disk for later use.

model.save('<path>/fashion.md')

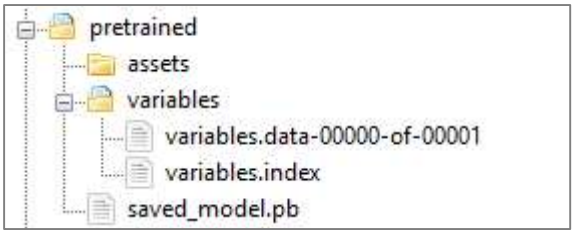


Figure 5: Directory Tree for saved Tensorflow model

At the time of this report, services that are dedicated to hosting AI models are getting traction, Tensorflow hub is a good example of such a service. In this portal

developers are able to publish their models for sharing with the community as well as consume some decent pretrained models. Actually, in the prototype computer vision apps I am using those pretrained models to test out the platform. Exactly at this time also, Facebook is teaming up with AWS to provide a model server for PyTorch library.

Step 2: Build the app

After the model is extracted and saved locally (or uploaded to a model server), the next step is to create code that pulls this saved model and runs the application. Since the platform is oblivious to how developers decide to implement their machine learning application, the interaction will be a simple method call with a set parameter and a return object that is generic enough to capture dynamic number of parameters and results.

The best way to model this interaction is a simple method signature that will accept a certain type of data and returns a certain type of data. This approach will allow the platform to spin up the app, pass a data structure to a method with a given name. Following this approach, the template will look like the snippet in Figure 6.

```
def run(data):
    #data['inputImage']
    #data['inputParameters']
     ....
    #process and prediction code goes here
     ....
    result = {
        #lable
        #image
    }
    return result
```

Figure 6: code snippet for a template run method

To account for additional utility modules that can be used in this method, as well as the set of resources (AI models, images or raw files) that go with it, the saved model in the previous step can be expanded to look like the figure below.

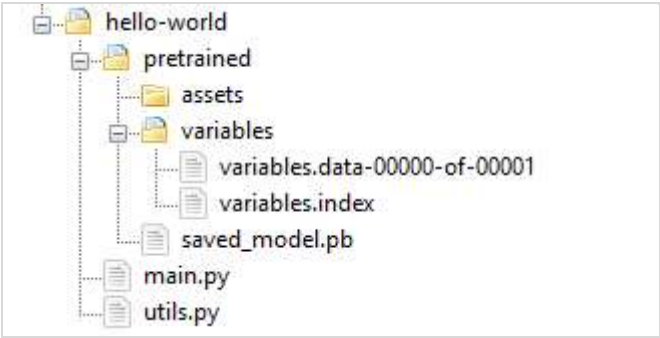


Figure 7: Directory Tree for a sample computer vision application package

In Figure 7 above, the main.py file is where the entry point to this app described in the method definition is found. this will be a typical content of a package that can be independently run.

In this step, developers would have a pretty clear idea of what is expected from users to do their processing. If the model needs more than just one image or additional textual or numeric data to supplement their prediction. and on the result end also, the developer would have a clear picture of what to display to the user as a result.

Step 3: Test and publish

In order to optimize and simplify the process of uploading source code with numerous supporting files, developers then will create an archive(zip) file out of these collections and upload the zip file all together. This step would just allow developers to identify the expected inputs, the expected result and most importantly they will upload the

zipped package that contains everything the platform, in this case Tensorflow, needs to run the application.

Developers will get an opportunity to test drive their application before releasing it to the general public by providing their own input with a known expectation.

RUNNING THE APPLICATION PACKAGE

After developers are done publishing their application, it is the platform's responsibility to load the package in the right environment with all the prerequisites installed and execute the "run" method with data provided by users from their devices. This infrastructure would need to meet these technical requirements at minimum:

- Prepare an environment which is granular enough to encompass the developer's environment when it comes to tools installed (a server that has all commonly used tools already installed on it).
- Load python source file in the application package dynamically as a module

The second requirement, that is, dynamically loading python modules and running them, is very integral to the success of the platform. python does allow (as a language feature) to import modules directly from an external source that is not part of the running code directory structure as well as not part of the local installed package repository. but this technical requirement will be at the top of the checklist when planning to support any other language in this platform. Python right out of the box provides a library called "importlib" that exposes the actual import infrastructure that runs python library linking.

There were some roadblocks that are specific to python 3.7 and Tensorflow, that I had to work through and find a solution, I have added documentation about this in the result section of this report. But the process of trying to sort out these workarounds was hampered by the fact that there was no direct access to the stack trace (exception dump), in the context

of running the computer vision app, when these syntax or runtime errors occur. This struggle highlighted the need for some kind of logging infrastructure that can broadcast monitoring and logging information to an area that is viewable to the developer.

Aside from the relatively small detours that were needed to make things working in python, the overall technical solution is actually generalizable and can be designed as a blueprint to support more tools and libraries. We can formulate a generic data structure and API template based on the walk through in the previous section.

COMPUTER VISION APPLICATION DATA STRUCTURE

This is a data structure that is used by the platform to package a computer vision app with all the information needed to run in the engine. Below are the attributes that describe a computer vision app.

Name	Data Type	Description
Name	Text	a display name for the application, that will be used in search as well.
Description	Text	a documentation of what this parameter is, so that users can easily understand and enter the required information.
Category	Text	a category of what best describes the application, so that the platform can place the application appropriately to users.
Version	Text	a standard version number to keep track of what is currently available on the application.

Table 1: continued next page

AppParameters	AppParameter *(described below)*	This is a list of parameters that will be passed to the core engine when consumer users run the application
ResultVariables	Text List	This is a list of variable names that web/mobile UI expects after the app is executed.

ResultLayout	Text	The platform will provide a set of templates to use to display the result from application execution correctly, but developers also have a chance to design their own templates to layout the result correctly to the user. It will be a lightweight HTML template with the result variables embedded. A sample of this layout looks like below.
		`<p>`
		`<h1>{{result.classification}}</h1>`
		`</p>`
		`<img src="{{result.accuracyPlot}}" />`

SourceCodeFile	application package	This is the file that will be used by developers to package their machine language source code that is independently runnable on a different environment. Using a standard archive file type makes a lot of sense here, since it means it will be easier to group required files (code and pretrained model files) and it will actually compress the file size for efficiency. I have dedicated a section for this component, as it is the meat of this report.

Table 1: Data definition for Computer Vision Application

Name	DataType	Description
ParameterType	Enum	This value is used by the web/mobile UI to pick the right type of input component to the user and enforce type level restriction on inputs. for example, the number parameter will be rendered as “number only textboxes”. In this prototype I have made these parameter types available: • Number • Text • Date • Image
Name	Text	This value is used by the backend infrastructure as a key value when users send the data via application execution.
Label	Text	This value is the text label used when the input is displayed to users.
Description	Text	This is the help text displayed to the user to guide then to enter the correct information.

Table 2: Data definition for computer vision application parameter

```
[
        {
                "ParameterType": "Image",
                "Name": "Image",
                "Label": "Image",
                "Description": "Upload or take a picture of your object"
        },
        {
                "ParameterType": "Number",
                "Name": "Age",
                "Label": "Age",
                "Description": "Enter the age of your object"
        }
]
```

Figure 8: Sample JSON for computer vision application parameter

COMPUTER VISION APP RUNNER

As described in the section "Running the application package", the platform will provide an infrastructure that can load computer vision applications in a preconfigured environment (python + Tensorflow in that example). This infrastructure can be wrapped into a pluggable component, with an API exposing the expectations, when it is interacting with the rest of the components of the platform.

This approach follows a factory design pattern, where the platform can easily expand the number and kinds of computer vision applications that it can support. All a runner has to do is provide end points that are compatible with the specs provided in the coming publish section.

The platform employs the standard HTTP Web API protocol to structure what is needed to make a runner be able to take on running a computer vision application package. This protocol is widely adopted and very easy to build. With this additional requirement, the runner not only has to execute the contents of the package, but they will also have to host a web API to make their capabilities available for the other components of the platform. The runner's primary will support two endpoints as described below.

Publish

This endpoint will be solely responsible for taking developers' application package and persisting it in a way that is runnable in the future. It will also be responsible for validating the content of the package and giving feedback to the caller, if it has been successful or not. The full detail of the endpoint with all the expectations will look like this.

- **URL**
  <baseurl>\publish
- **Method**
  POST
- **URL Params**
  None

- **Data Params**
  CvappId = [Integer]
  SourceFile = [Zipped File]
- **Success Response**
  Code: 200
  Content: { hasError: false }
- **Error Response**
  - Code: 400
    Content: { message : text }
  - Code: 500
    Content: { message : text }

Run

This endpoint will be used to accept inputs from users and run the requested computer vision application and return the result in the expected format. The full detail of the endpoint with all the expectations will look like this.

- **URL**
  <baseurl>\run
- **Method**
  POST
- **URL Params**
  cvappid=[Integer]
- **Data Params**
  data = [array]
- **Success Response**
  Code: 200
  Content: { result: {} }
- **Error Response**
  - Code: 400
    Content: { message : text }
  - Code: 500
    Content: { message : text }

By following this spec, the platform can support many more tools that are not discussed in this report or even that are not even publicly available at this time.

Securing user inputs

As with just about any application that processes user information, it is very important to ensure the security of the data collected from users. It is even more pronounced, when it comes to capturing users' pictures. For this platform to make any impact as a machine learning tool for the general public, it must make sure that user's information is kept secure. There are three aspects of this requirement that will be handled by the platform.

1. Data security at rest
2. Data security in transit
3. Computer vision application data access control

The platform will utilize proven and industry standard methods to ensure data security at rest and in transit. This is particularly built to stop hackers from stealing users' information. Credentials will be encrypted, and user images will be kept on servers that are only available for authenticated users. There are a variety of mature solutions readily available to achieve this, and this report won't go in detail about them. But the main concern that is rather unique to this platform is the potential for invasion of privacy of users by exposing their photos for untended machine learning algorithms. There will be plenty of access control measures to make sure user's data is kept away from unauthorized processing.

Images uploaded as an input to computer vision application will be uploaded to the general platform storage but a unique image URL only identifiable by the target computer vision application will be passed to the computer vision application runtime.

Computer vision application runners will be firewall locked (outbound) to communicate only to the web servers, via a specific IP address, port and protocol. This is

to ensure that the applications won't circumvent the platform and start taking data away from the platform. The web application will include a feature to ask for consent to allow computer vision applications to take advantage of users' inputs to further fine tune their models

Prototype Setup

Based on the design and architecture I outlined in the previous sections, the prototype for the platform is built using the easily accessible cloud infrastructure these days. The prototype uses a mix of AWS and Azure just to test out the componentized approach that was laid out in the design section.

The user interface for the platform is built using asp.net core using the MVC infrastructure available. I have also used AngularJS and Typescript as the client-side scripting language. The application user interface is responsive where it is easily accessible on both desktop and mobile devices. Due to time limitations, I was forced to cut down on the features I envisioned to include in the prototype as researching the tools especially the version segmentation that comes with python 2, 3, 3.7 etc took a large chunk of my time.

For file storage I used the file system of the servers themselves, but that is one of the areas of the platform that needs to be moved to a service that is more scalable and secure. The resources used to run tests on the platform are list below:

- Web Application Server
    - Azure App Service
        - OS: Windows Server 2016 HyperV
    - Web server: asp.net core
    - Standard S1
        - 1 Core
        - 1.75 GB RAM
        - 50 GB Disk Space
- Tensorflow Runner

    - ec2 - CPU
        - OS: Ubuntu Server 18.04 LTS
        - Web server: apache2
        - c5.2xlarge
            - 8 vCPU
            - 16 GB RAM
    - ec2 - GPU
        - OS: Ubuntu Server 18.04 LTS
        - Web server: apache2
        - p2.xlarge
            - 1 GPU
            - 4 vCPU
            - 61 GB Memory
            - 2496 Parallel Processing Cores
            - 12 GB GPU Memory
- Database
    - AWS RDS
        - SQL Server Express Edition

SAMPLE APPLICATIONS

I used existing sample code to create the computer vision applications needed for benchmarking. I had to do minor refactoring so as to make the codes to meet the API requirement of the runners. The list below shows the sample applications published to the prototype portal.

1. **Object Detection**: This is a sample code ported from Tensorflow tutorial, and its output will do object detection on a given image and returns a result containing the objects detected in boxes with their corresponding score.
2. **BigGAN**: This is also a sample code that utilizes a BigGAN model available on Tensorflow hub. And what it does is generates a high-resolution diverse sample for a given category.

3. **Insect Classifier**: This is a sample code that utilizes an insect classifier model available on Tensorflow hub. After processing an image, it returns an insect classification label from a list of around 1020 classes.
4. **Plant Classifier**: This is a sample code that utilizes a plant classifier model available on Tensorflow hub. After processing an image, it returns a plant classification label from a list of around 2100 classes.
5. **Stylizer**: This is a sample code that uses an arbitrary image stylization model from Tensorflow hub and applies a visual style to an input image to generate an artistic version of that image. at the time of this prototyping, the application only applies the same style on a given image.

RESULT

Tooling

At the beginning of this report one of the goals was to research the availability of out of the box tools that can be easily plugged into the platform I described thus far. But the research narrowed down to exploring one of the most popular languages for machine learning (Python) and Experimenting with Tensorflow as one of the tools that will be supported by the platform.

I found Python to be very versatile, easy to pick up and very powerful. But the different versions that are available to use (2. * and 3. *) and how each python library is always playing catch up with the changes being introduced in each version upgrade was time consuming to manage. That fact also made documentation for any of the libraries to be quickly outdated and obsolete in a matter of months. This will be particularly challenging for a platform that is going to be accepting source codes.

I was able to explore setting up a runner infrastructure for Tensorflow 2, with a web API endpoint and it was quite challenging to get everything to work right out of the box. These are some of the challenges I had to work through.

- Flask and Keras python libraries don't work well together in production settings in Flask. I kept getting run time errors originating from Keras when I tried running my Tensorflow code that had Keras library reference. After some research, I found out that at this time, it is a known issue and can be resolved by running Flask in debug mode.
- When loading python modules using the importlib utility, the current directory stays the same for the loaded module. This meant, any relative path reference will not work right off the bat. At this time, I was able to get past this roadblock by

utilizing the current location of the module itself instead of relying on the relative pathing.

- The most time-consuming part for me was trying to piece together python libraries that will work well with the latest version of Tensorflow (2.2.0). It took numerous frustrating nights to allow a simple CPU bound Tensorflow computer vision application to run in a Flask API that is hosted in Apache through mod_wsgi.

I had to rely on release candidate Tensorflow 2 versions as well as building from source in order to accommodate python 3.7 use with web API library. But overall, I felt most of these issues might be related to my lack of experience using the tools and language and are truly business as usual for an individual in this industry.

Performance and cost

I created two copies of the Tensorflow computer vision application runner that utilizes CPU and GPU to observe differences in the performance. Below is the result for response time and cost associated with each sample application.

After running 100 run requests per sample computer vision application per machine, there were a couple of interesting observations. In order to clearly illustrate how the actual machine learning processing is, the full execution is broken into individual checkpoints for dynamically loading the application source, loading model (downloading from a repository and loading in memory), running the prediction and generating the result.

Table 3 shows the average time in milliseconds it took to execute each step from loading the application in memory to generating the result.

Computer Vision Application	CPU						
	App Loading	Pre-Processing	Loading Model	Running Model	Generating Result	Misc	Total
Insect Classifier	174	260	2,049	681	195	200	3,559
Object Detection	232	177	17,598	6,870	514	483	25,874
Plant Classification	226	409	2,203	770	236	252	4,096
Stylize	311	289	2,502	2,577	265	464	6,408
BigGAN	842	162	14,988	27,856	530	676	45,054

Table 3: Average execution time in milliseconds for each sample applications running in general purpose server ec2-CPU

Initially the long response time, more than 3 second for the simplest classifier, from the runners was not motivational to look at. But after breaking down the execution to see where the runner spent most of its execution time, it can be easily seen that the prediction is actually not the primary source of processing consumption.

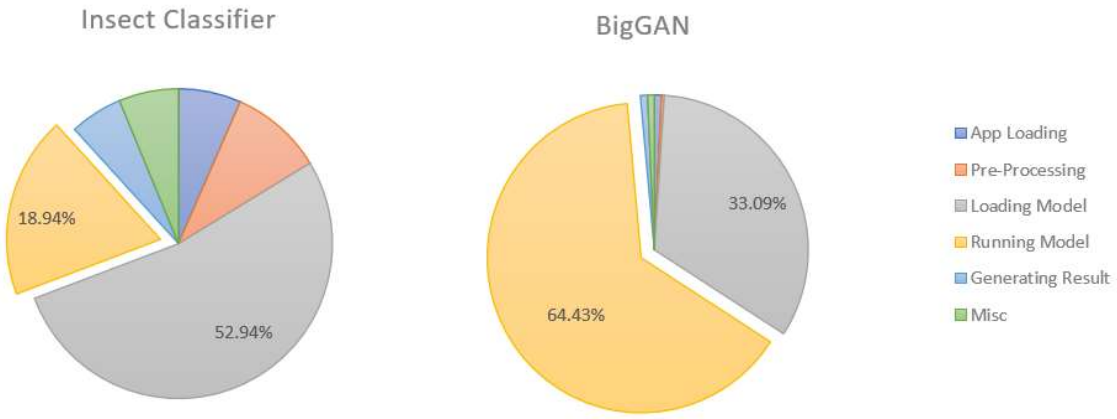


Figure 8: Proportion of machine learning processing versus the other application execution

The pie chart in the figure above shows how the execution time was distributed for the simplest (Insect Classifier) and most complex (BigGAN) sample applications. In both cases loading the model to memory took a sizable amount of time. But running time in BigGAN was the longest, it took an average 64% of the processing time.

Computer Vision Application	GPU						
	App Loading	Pre-Processing	Loading Model	Running Model	Generating Result	Misc	Total
Insect Classifier	210	311	1,691	605	177	200	3,194
Object Detection	465	168	13,607	5,487	418	1,156	21,301
Plant Classification	191	445	1,741	702	175	168	3,422
Stylize	281	251	1,798	1,294	173	449	4,246
BigGAN	491	179	21,394	41,659	484	452	64,659

Table 4: Average execution time in milliseconds for each sample applications running in GPU instance ec2-GPU

Running these sample applications on a GPU instance yielded a boost in performance (not to a significant extent). The trend below shows the comparison between machine learning bound processing times for the sample applications.

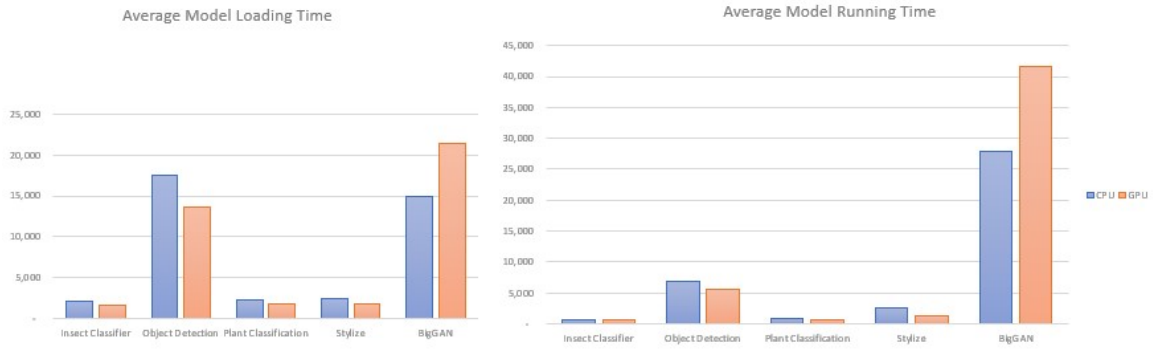


Figure 9: Average Model loading and running comparison between CPU and GPU runners.

For the simpler model, except for BigGAN, the runner on the GPU was slightly better than the ones in CPU. But the BigGAN sample application CPU was better in both loading and running the model. That result goes against the expectation that the GPU will perform better than the CPU all the time. Although the performance difference was not as stark as I expected, there was one observation that made the GPU machine stand out from the CPU machine. That the CPU machine was unable to handle having loads of concurrent

requests but the GPU machine was able to handle concurrent requests with similar performance.

The cost associated with hosting these computer vision application runners was $0.34 per hour for the CPU machine and $0.90 per hour for the GPU machine.

CONCLUSIONS

The results of the prototype are very encouraging from a feature standpoint as well as performance. Tensorflow was widely available to install as a standalone package and works with other libraries to orchestrate machine learning capability from a web user interface. Packaging pretrained models, saving, loading and executing in all of the major machine learning libraries is provided out of the box. And hence with some technical taming, the current state of machine learning tools is ripe for such a platform to flourish.

As the result from performance testing shows, the execution time for a simple and targeted machine learning model is small and the response time for runners to perform a prediction can be further optimized by addressing the other latencies in the pipeline. Caching models in memory would eliminate the time it takes to load a model every time a request is made. And the same goes with dynamically loading the application package, by caching that imported module, and then the runners can speed up the overall response time.

This experiment has shown the tools needed to build such a platform are readily available and the performance that can be attained with minimal cost is also feasible.

FUTURE WORK

This report is just the skeleton of what is possible in delivering computer vision capabilities to the general public. From expanding the feature set provided by the platform to supporting more runners, the complete implementation of this platform will follow beyond this report.

On top of the must have features that are listed in the list of features section of this report, adding features for the platform to work more like a mobile app store is on top of the feature expansion vision.

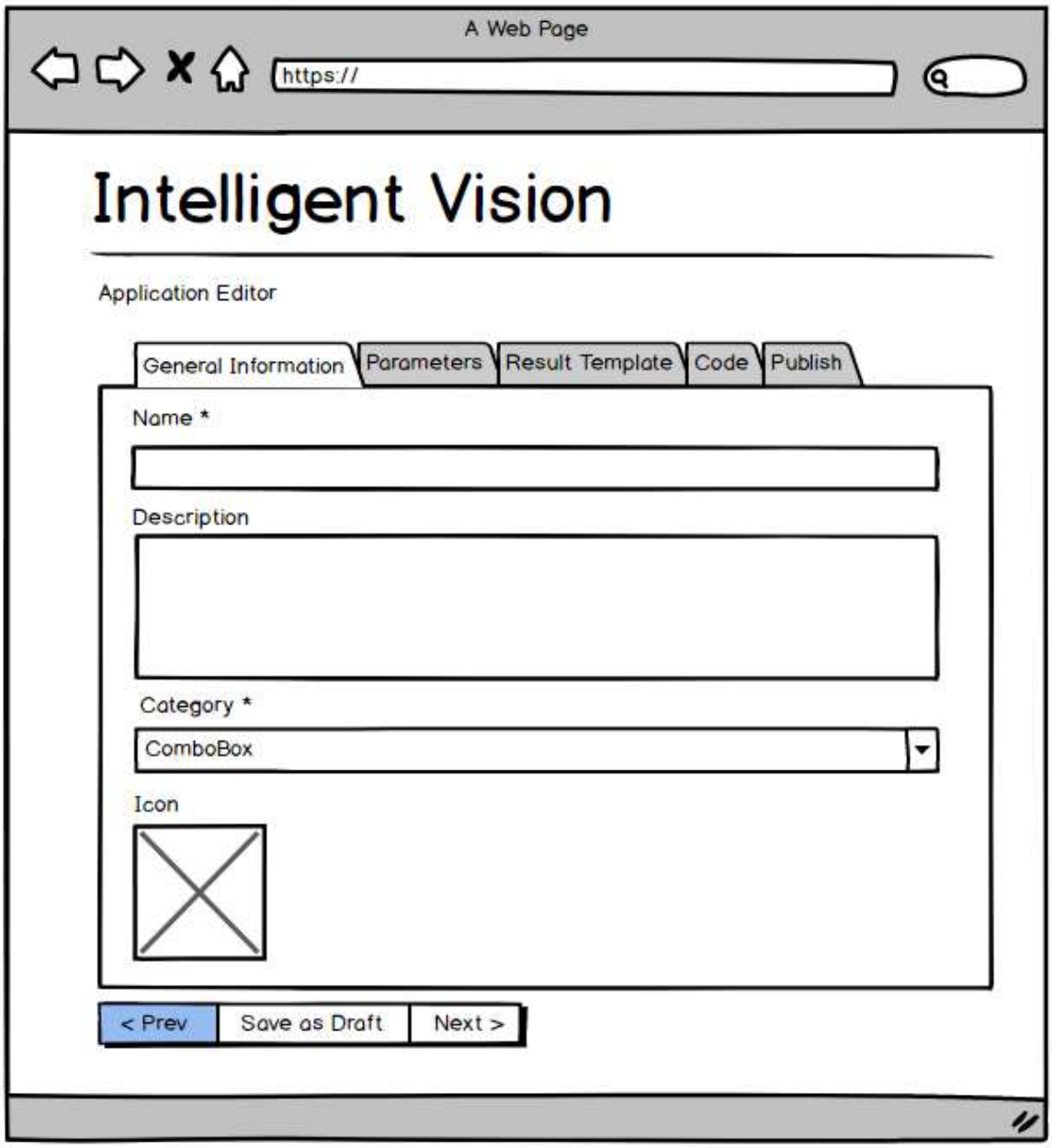
- Ability for developers to provide paid computer vision applications
- Ability for the general public to search for computer vision application developers for their unique needs.
- Introducing app review process for quality and security
- Providing runners that vary in computation power for developer to choose to publish on

Features like these would greatly incentivize developers and establish a sustainable foundation for the platform to mature. Expanding support for more machine learning libraries and languages will also attract more developers to host their applications on the platform. And clearly Pytorch has to be the next library to support as a runner.

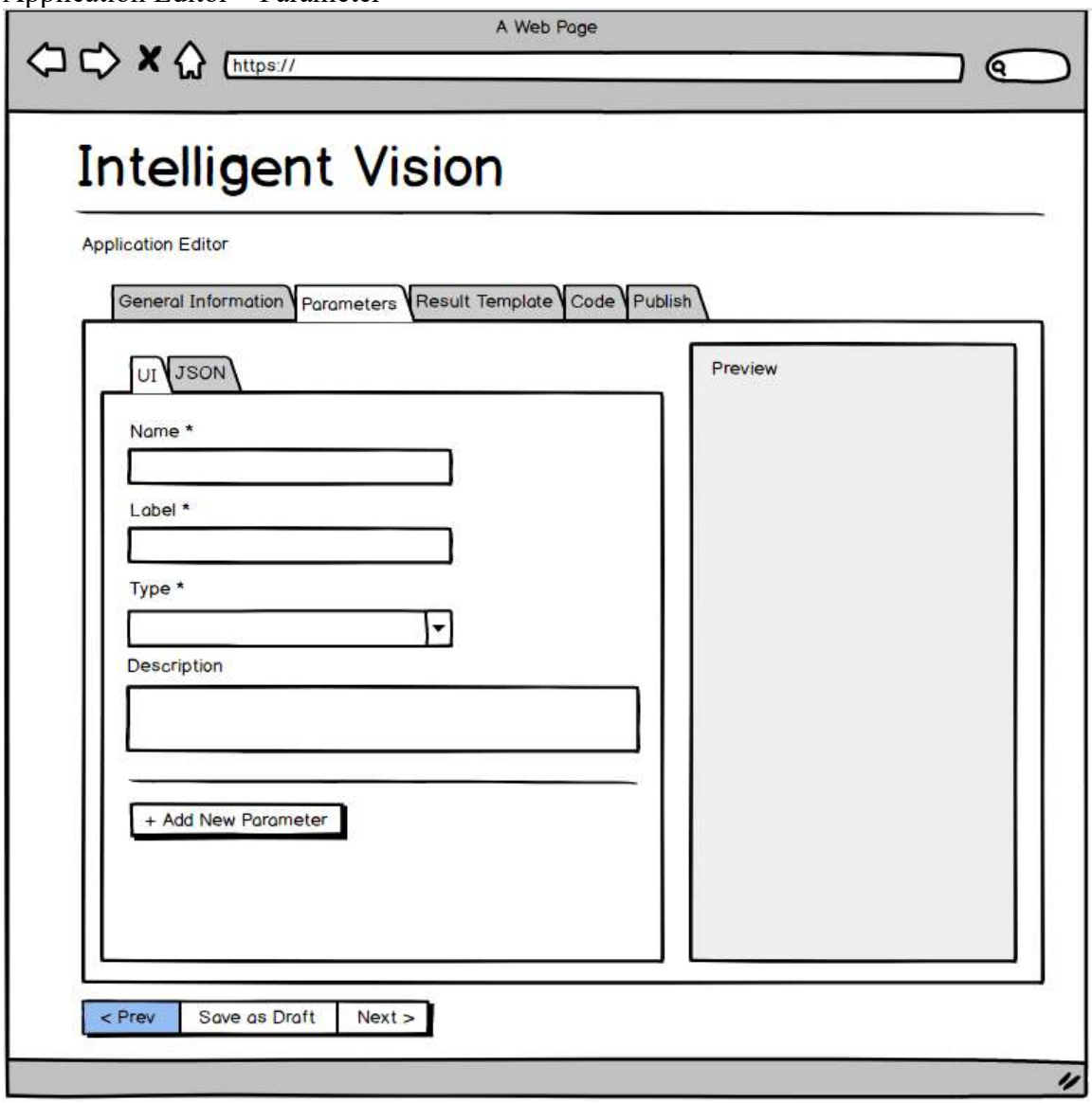
Appendix

Wireframes

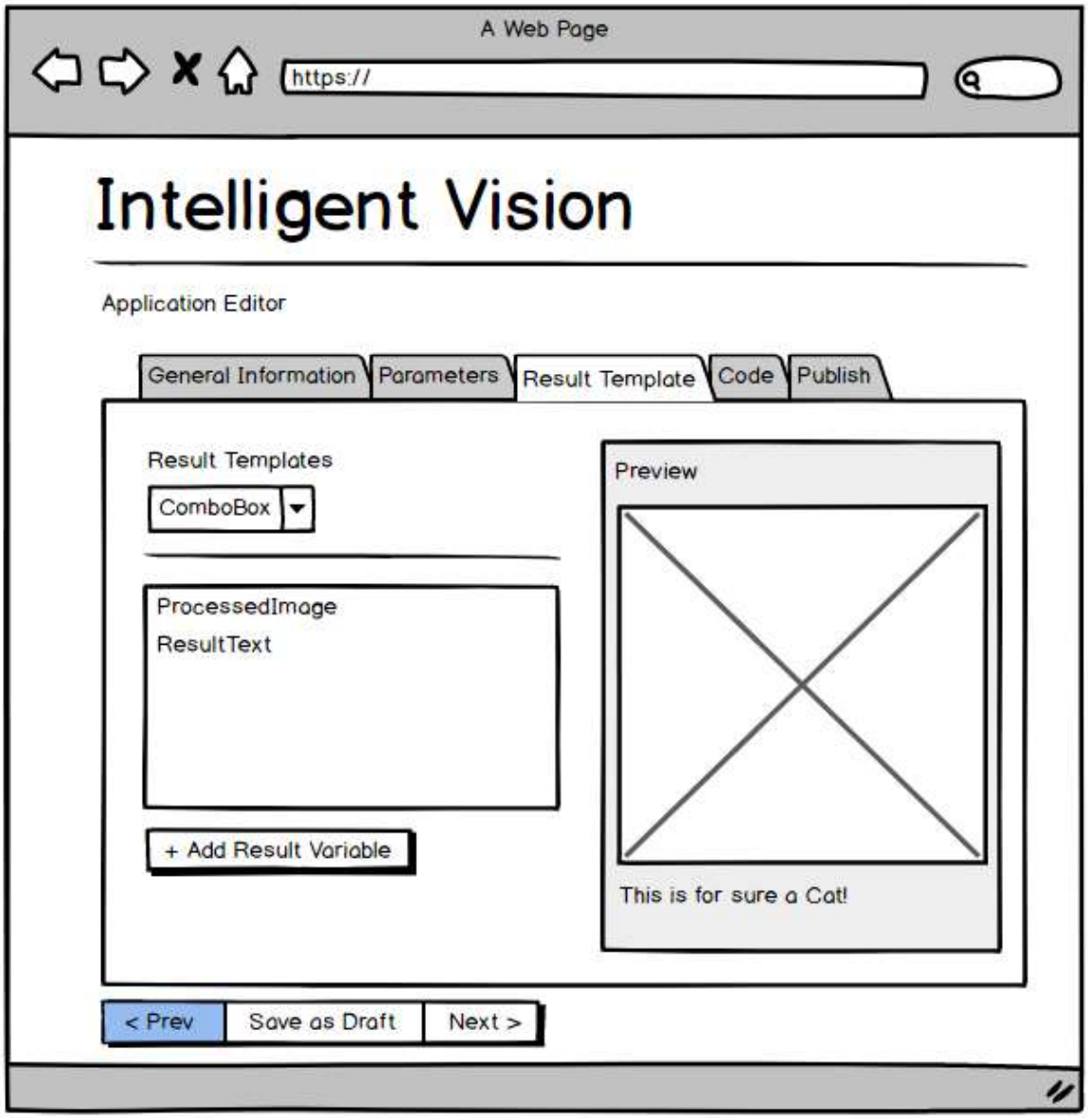
Application Editor - General Information



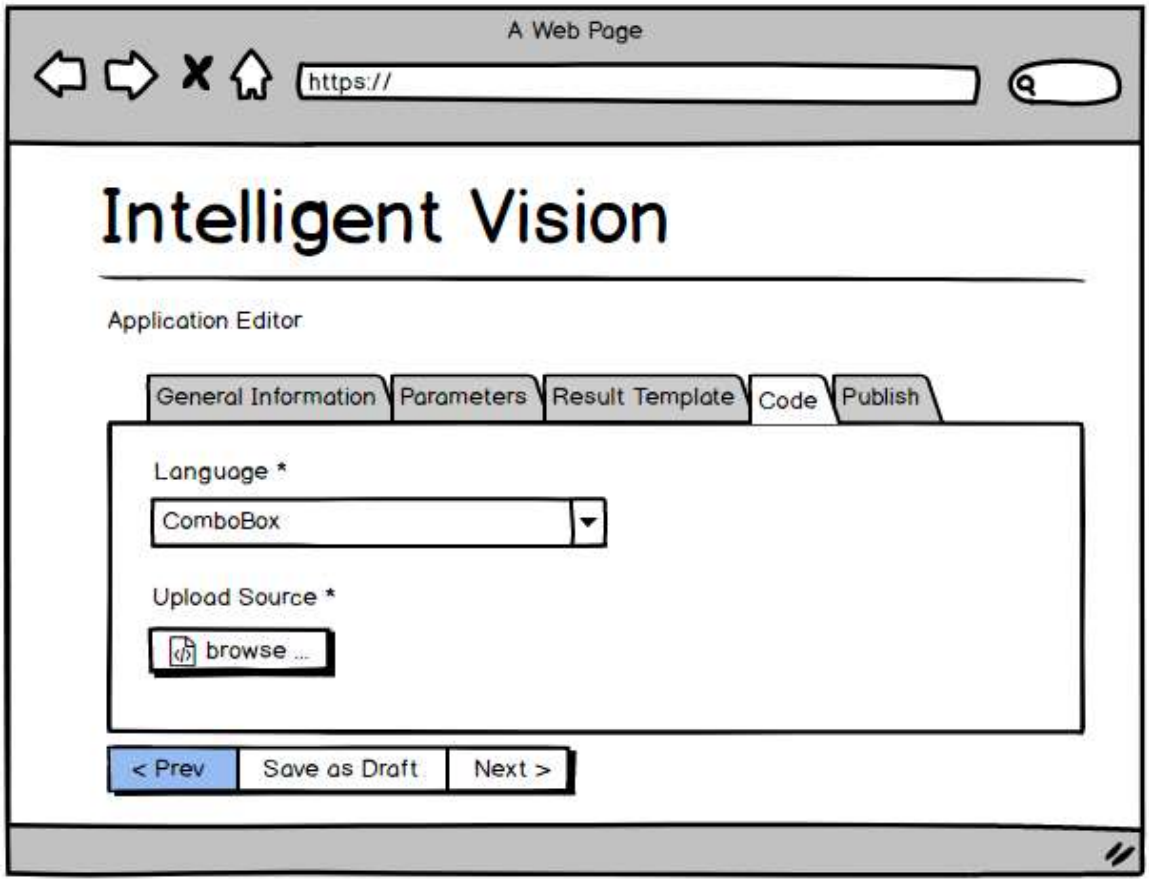
Application Editor – Parameter



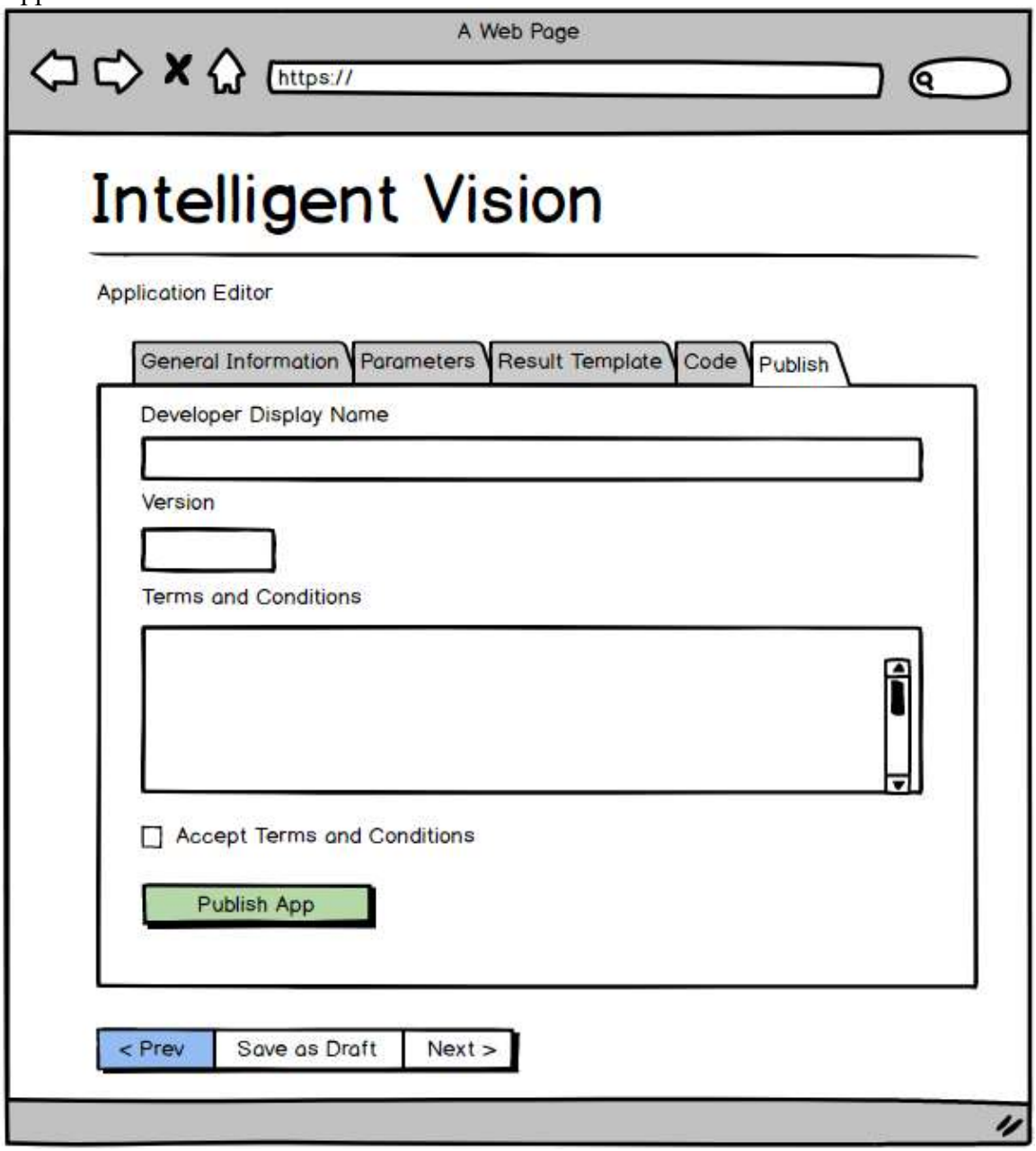
Application Editor - Result Template



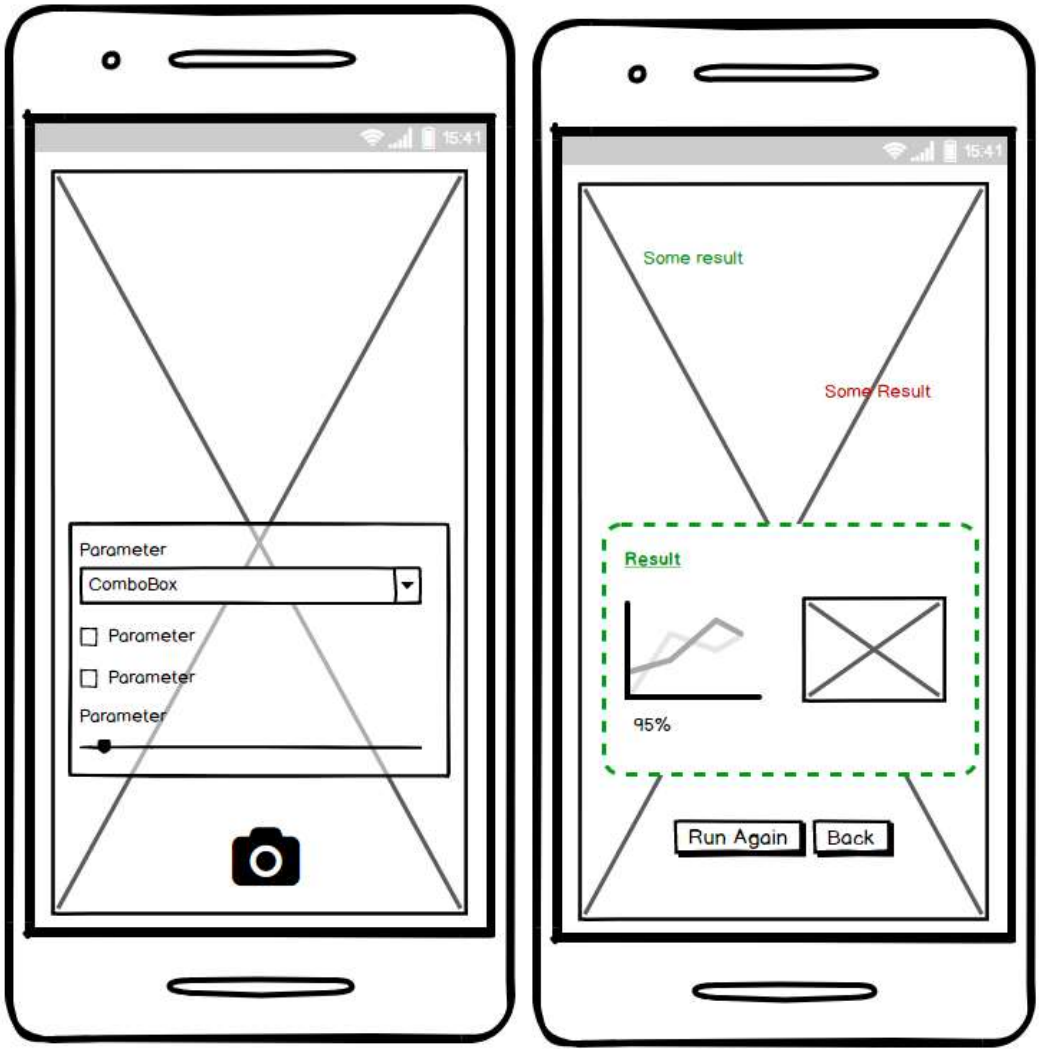
Application Editor – Code



Application Editor – Publish



Mobile Application Browser



References

Arianna Dorschel, Rethinking Data Privacy: The Impact of Machine Learning, Apr 24, 2019. https://medium.com/luminovo/data-privacy-in-machine-learning-a-technical-deep-dive-f7f0365b1d60

Manish Prabhu, Security & Privacy considerations in Artificial Intelligence & Machine Learning — Part-6: Up close with Privacy, Feb 8, 2019. https://towardsdatascience.com/security-privacy-in-artificial-intelligence-machine-learning-part-6-up-close-with-privacy-3ae5334d4d4b

Fashion-mnist sample classifier, https://github.com/zalandoresearch/fashion-mnist

Tensorflow documentation and samples https://www.tensorflow.org/

Augment Reality Sample Picture https://tech.wayfair.com/tag/augmented-reality/

Web API specification https://en.wikipedia.org/wiki/Web_API

Flask python web API library reference https://flask.palletsprojects.com/en/1.1.x/

Large Scale GAN Training for High Fidelity Natural Image Synthesis

Andrew Brock, Jeff Donahue, Karen Simonyan https://arxiv.org/abs/1809.11096

Arbitrary Image Stylization https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2

Tensorflow active issue reference https://github.com/tensorflow/tensorflow/issues/34607